Estudiante: Christian Porfirio Cevallos Armijos

Asignatura: Lógica de Programación

Fecha de elaboración: 19 de agosto de 2025

Evaluación en Contacto con el docente

Tema: Impacto de las nuevas tecnologías en la sociedad: visualización del futuro.

Informe de Análisis: Implementación y Gestión de Estados en el Juego de la Serpiente

INTRODUCCIÓN.

El presente documento analiza el código del juego "Snake" implementado en Python utilizando la biblioteca Pygame. El enfoque principal se centra en la gestión de estados del juego y cómo esta estructura influye en la experiencia del usuario, la claridad del código y la escalabilidad del proyecto. A través de un examen detallado, se exploran las decisiones de diseño, las ventajas de la implementación actual y las oportunidades de mejora, con el objetivo de entender cómo los sistemas de estados bien definidos pueden enriquecer proyectos de desarrollo de software, especialmente en el ámbito de los videojuegos.

ANÁLISIS DE LA GESTIÓN DE ESTADOS.

1. Estructura de Estados Implementada.

El código utiliza un sistema de estados finitos para gestionar las diferentes fases del juego, definidas en el diccionario ESTADOS_JUEGO. Esta implementación organiza el flujo del juego en cuatro estados claramente diferenciados:

- **JUGANDO (0)**: Estado principal donde ocurre la acción del juego
- **PAUSA (1)**: Modo de interrupción temporal que mantiene el estado actual
- **GAME_OVER (2)**: Estado final cuando el jugador pierde
- **PUNTUACIONES_ALTAS (3)**: Visualización de los mejores resultados

Esta estructura me parece particularmente acertada, ya que proporciona una separación clara de responsabilidades y facilita la navegación entre las diferentes pantallas del juego. Me llamó la atención cómo el autor evita el uso de variables booleanas múltiples (como pausa_active, game_over_active, etc.) que suelen complicar la lógica en implementaciones menos sofisticadas.

2. Transiciones entre Estados.

Las transiciones entre estados se gestionan principalmente a través de la función manejar_eventos(), que procesa las entradas del usuario y determina los cambios de estado correspondientes. He notado que esta centralización del manejo de eventos es

un acierto importante, ya que evita la duplicación de código y hace que el flujo del juego sea más predecible.

Un detalle que considero especialmente elegante es cómo se manejan las direcciones opuestas mediante el diccionario DIRECCIONES_OPUESTAS, que previene movimientos inválidos (como que la serpiente se dirija en sentido contrario instantáneamente). Esta implementación demuestra una comprensión profunda de la mecánica clásica del juego.

3. **Persistencia de Datos y Puntuaciones.**

El sistema de guardado de puntuaciones utilizando JSON es funcional y adecuado para el alcance del proyecto. Me parece interesante cómo el autor implementó las funciones cargar_puntuaciones() y guardar_puntuaciones() de forma independiente, lo que sigue el principio de responsabilidad única. Sin embargo, noté que la función actualizar_puntuaciones_altas() podría optimizarse para evitar cargar y guardar el archivo completo repetidamente en sesiones prolongadas.

PATRONES Y TÉCNICAS DE IMPLEMENTACIÓN.

1. **Organización del Código.**

El código muestra una clara influencia de principios de diseño como DRY (Don't Repeat Yourself) y separación de concerns. Las constantes se agrupan en diccionarios temáticos (COLORES, DIRECCIONES, etc.), lo que mejora significativamente la mantenibilidad. Me gustó especialmente cómo se utilizan las comprensiones de listas en funciones como generar_comida(), donde se crea una lista de posiciones libres de manera eficiente.

2. **Renderizado y Separación de Lógica.**

La separación entre la lógica del juego y el renderizado es otro aspecto destacable.

Funciones como dibujar_serpiente(), dibujar_comida() y mostrar_puntuacion() mantienen una clara división entre el estado del juego y su representación visual. Esto facilitaría enormemente implementar, por ejemplo, un cambio de tema visual sin afectar la lógica principal.

3. **Manejo de Colisiones.**

La función verificar_colision() implementa de manera eficiente dos tipos de detección: con los bordes del tablero y con el propio cuerpo de la serpiente. Me pareció inteligente el uso de serpiente[:-1] para omitir la cola en la verificación, ya que este segmento se moverá en el mismo frame.

LIMITACIONES Y OPORTUNIDADES DE MEJORA.

1. **Acoplamiento en la Función Principal.**

La función main() muestra cierto grado de acoplamiento, manejando directamente las transiciones de estado y la lógica del juego. En mi opinión, una implementación basada en clases podría mejorar la organización, aunque entiendo que para un proyecto de esta escala la aproximación procedural es perfectamente válida.

2. **Gestión de Recursos y Rendimiento.**

El juego regenera todas las posiciones libres cada vez que necesita crear nueva comida, lo cual podría volverse ineficiente en tableros muy grandes o con serpientes muy largas. Una alternativa sería mantener un conjunto actualizado de posiciones libres que se actualice incrementalmente.

3. **Internacionalización y Accesibilidad.**

El código contiene texto hardcodeado en español, lo que limita su accesibilidad a jugadores de otros idiomas. Además, no se observan consideraciones para jugadores con discapacidades visuales (como opciones de alto contraste o tamaño de texto ajustable).

4. **Implicaciones y Aplicaciones Más Allá del Juego.**

El sistema de estados implementado en este juego tiene implicaciones interesantes para el desarrollo de software en general. La clara separación entre estados, transiciones y acciones es aplicable a múltiples dominios, desde interfaces de usuario hasta sistemas de control industrial.

5. **Escalabilidad del Sistema de Estados.**

Si bien el sistema actual funciona adecuadamente para este juego, noté que podría volverse cumbersome si se agregaran más estados (como menús de configuración,

selección de dificultad, o pantallas de tutorial). Una implementación alternativa utilizando una máquina de estados finitos (FSM) más formal podría mejorar la escalabilidad.

En mi experiencia desarrollando proyectos académicos, he notado que los estudiantes (incluyéndome) tendemos a subestimar la importancia de una gestión de estados bien planificada, lo que often resulta en código espagueti difícil de mantener. Este proyecto sirve como un excelente ejemplo de cómo implementar este concepto de manera efectiva.

La técnica de usar diccionarios para mapear constantes (como DIRECCIONES y DIRECCIONES_OPUESTAS) es particularmente valiosa, ya que reduce errores por tipos y hace el código más legible. Planeo incorporar este enfoque en mis futuros proyectos.

CONCLUSIÓN.

El análisis del código del juego de la serpiente revela una implementación sólida y bien estructurada, con un sistema de estados que efectivamente gestiona las diferentes fases del juego. La separación de concerns, el manejo eficiente de colisiones y la persistencia de datos demuestran una comprensión avanzada de los principios de programación, especialmente notable considerando que se trata de un proyecto académico.

Las principales fortalezas residen en la claridad del código y la gestión organizada de los estados del juego, mientras que las oportunidades de mejora se centran en la escalabilidad y optimización de ciertas funciones. Este proyecto ilustra cómo incluso en implementaciones aparentemente simples, las decisiones de diseño tienen un impacto significativo en la mantenibilidad y extensibilidad del código.

La extrapolación de estos principios a otros dominios del desarrollo de software es directa: la gestión adecuada de estados es crucial en cualquier sistema complejo, y las técnicas aquí demostradas (especialmente el uso de diccionarios para mapeo y la centralización del manejo de transiciones) pueden aplicarse ventajosamente en proyectos que van mucho más allá del desarrollo de juegos.

Este análisis me ha hecho reflexionar sobre la importancia de diseñar sistemas con una arquitectura clara desde las etapas iniciales, incluso en proyectos que parecen sencillos en superficie. La inversión en una base de código bien estructurada paga dividendos significativos cuando llega el momento de extender o modificar la funcionalidad.